Checking CFD interfaces in a multi-disciplinary workflow with an XML/CGNS compiler

M.Poinot [*], E.Montreuil [†], E. Henaux [‡]
ONERA, Chatillon, F-92322, France

Multi-disciplinary CFD computations are involving many kind of solvers, pre and post processors. Some of these components may be third party software, proprietary and licensed software or even something which does not exist yet. The CGNS standard allows clear and well-defined interfaces between these CFD components to be defined by the users. The specification of such interfaces are done with XML, a text based tree representation. The conformance of a given interface, or a given set of data produced or consumed by a component can be achieved using the XML/CGNS compiler. An example in the area of icing simulation is presented.

I. Introduction

As the knowledge of physics and the powerfulness of computer grow, the simulations are getting more complex, involving multi-disciplinary codes and big amount of data. Instead of building large and all purpose code, the assembly of code components into a single process is a more versatile way to obtain a dedicated application. Most of these code components can be *COTS*[1] (*commercial off the shelf*), a proprietary code or a new component written for a specific purpose. All components can be tied together using a *steering programming language*.[2] This assembly process of selected components may lead to use a component for which only the interface is known, not the inside. Such an approach is also known as the *black-box* approach. You have knowledge of the *public* information provided by the component, but you don't know how it is done. You have to trust the *black-box*, and eventually to test it to make sure it conforms to your application requirements.

In order to select components and to check their interfaces, we are using standards and we have developed a tool to check actual *interoperability* of components. The public specification of these interfaces is made using a textual representation of the CGNS[3–5] standard structures. The CGNS standard is a good candidate for the CFD workflow interface definitions.[6] Each component can specialize its own specification, as far as this latter is a subset of the original CGNS SIDS[a] specification.[7] The process of submitting, reviewing and publishing the per-component specifications is made with all the participants of the CFD system.

We are presenting here an approach already used in other computer science areas, such as database systems or network systems.[8] A compliance check tool, that could be called a *CGNS compiler*, has been realized to achieve the verification of input/output data of a set of CFD workflow components. The compiler takes an XML[9] file as input, and produces a diagnostic and an HDF[10] binary file as output. The XML file is the textual representation of a CGNS tree, the HDF file is the actual binary *value* of this file.

We illustrate the use of this compiler with the example of a simulation workflow in the area of *icing* research.[11] The codes are developed and used by the teams working on the project. The use of the XML tree description for the input/output for every component has improved their work in terms of better definition of components. The use of the generated HDF files has increase the reception tests capabilities.

[*] Software Engineer, Computational Fluid Dynamics and Aeroacoustics, Member AIAA
[†] Research Scientist, Physics Instrumentation and Sensing, Member AIAA
[‡] Student, Computational Fluid Dynamics and Aeroacoustics, -
[a] SIDS stands for "Standard Interface Data Structure", it is the main CGNS document, already an *AIAA recommended practice* referenced *AIAA-R-101A* and in the process of being an *ISO* standard

II. The CFD workflow

A. Multi-components simulations

The simulations are now involving many solvers. The architecture of these systems are sequential or parallel executions of simulation codes, the control and the data exchange between codes are performed by dedicated applications mostly written with the so-called *steering languages*. An industrial or a research team does not write the whole system, but rather built a system re-using existing components. The *steering application* (see fig. 1) controls the calls to the components and give then the relevant parameters, insure data transfers when required or performs some data tuning (i.e. dimensional to non-dimensional data).

Building an application is a matter of *assembly* of components in order to obtain the expected algorithm. The *assembly team* may not be able to check all component contents but it should control the whole system behavior. And this behavior strongly depends on the consistency of the component interface.

This raises new issues, such as: *Is my output readable by your code?* or *How can I test your code which still is in development?* or *How can we foresee extensions for the next project phase?* And many others, because the user is not the owner of the whole system. The user must take into account the modification that eventually will occur because software components are always evolving. This is especially true in a research environment, which follows the leading edge of both physics and computer science. The component interface may not be well defined, or will change as the knowledge of the required input and output of this black-box grows.

Now your system is getting more robust. A component can be replaced by another one (for example a version of a software can be replaced by a new version), or a prototype can be used at development time and then replaced by an actual code. You are able to change these components if they have the same interface. This is not that easy, because in numerical simulation the whole system behavior (i.e. the global algorithm) is tightly bound to the components behaviors (i.e. the local algorithms). However, for some specific black-boxes with well defined interfaces, such a *component exchange* could be done. For example you can change boxes related to some mesh related modifications (i.e. deformation, remeshing, interpolation...) or tracking convergence, visualization. We actually have much more complex components, in that case we do *require* to have knowledge of the internal algorithms of the components. Every time we detect an *hidden* information which has a strong impact on the computation, we require to make it *public* when this is possible.

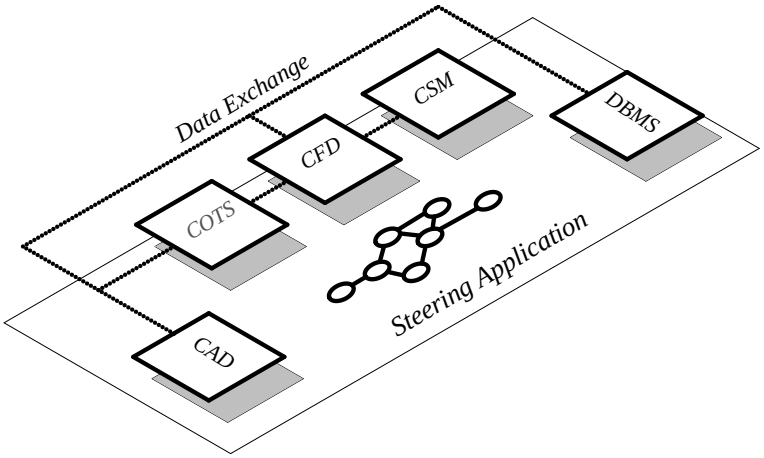

Figure 1. Multi-disciplinary simulation involving components

Taking apart the problem of defining algorithm and their dependencies[b] ,we focus on the interface in terms of a consistency set of data input and output.

[b]Algorithms can be defined using some *finite state machine (FSM)*. Most of the time, components are documented with their *API (application Programming Interface)* which does not contains the behavior automata (FSM), at least described in a standard or a formal form.

B. Components and CGNS interfaces

The *interoperability* between black-boxes is achieved through standard interfaces for the CFD simulation components. One may have no knowledge of *how* a computation is done, but rather *what* kind of computation is done, this introduces the concept of *interface*. This eventually leads to more and more modularity of the system, you build your simulation workflow by looking at interfaces. Some systems, such as CORBA[12] or even MPI, have clear and powerful means to define interfaces, but they are defining low level protocol for control and data exchange. We have to define more precisely our data, because the meaning of the data in a simulation is the *glue* that makes components a *system* when connected to each other. The more the interface is specialized, the better it is for the user. A specific interface allows a very high level of semantic embedded in the definition to be obtained by the user. For example, if your component declares to output the `TurbulentEnergyKinetic`, you immediately have a more precise knowledge of the data. The interface should take into account the context of the simulation.

In the area of CFD, we already have the CGNS specification. Even if this specification is not complete enough to insure all data description in a multi-physics simulation, it is extensible and immediately usable.

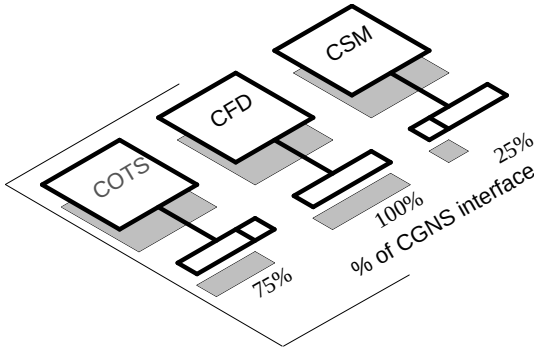

Figure 2. CGNS mapping of an interface in CFD workflow

Any kind of component should have a CGNS interface, as soon as this component can be selected for a CFD workflow. It means that if this component is a good candidate to be used in a CFD simulation, then there is at least a minimal CGNS interface for it. As shown in figure 2, a component can provide only 10% of the CGNS interface.

C. Conformance to standard

Once a standard, or a set of standards have been defined, all black-boxes can be selected, designed or adapted to fit the requirements. Then a new step comes in our assembly process: we have to check the conformance of the black boxes with respect to the standards. In the case of third party software, you can trust software providers because they all use the so-called *mid-level library* which enforces the checks. This actually is the only way to read or write a CGNS binary file and to check the correctness of a data structure. If the library calls do not complain at the creation time, then the user can believe its tree is correct.

The next CGNS version will use *HDF5* storage layer, the software providers can now directly use this public library, and some could make some *non-CGNS-compliant* extensions or modifications. We require a mean to check a CGNS interface. Such a mean should be composed of a language for the interface specification and tools[c] to check the interface conformance with respect to the SIDS.

We already have a grammar which allows to define a CGNS data structure and a CGNS data value. However this grammar is almost dedicated to SIDS and we cannot accept to make some new tools using this proprietary grammar and at the same time claim we have an *Open System*. We had to select a textual representation that could be understood and modifiable by all CFD users. We decided to enter into the *XML world*, which publishes *open standards* and which provides many tools and documentation.

In terms of standard compliance, the fact of being CGNS compliant, for a component, can lead to a label. So far, there is no *CGNS compliant* label, but certainly this is a task that should be done by consensual groups such as the *CGNS steering Committee*.

[c] The more we have compliance tools, the better it is.

III. Interface specification and conformance check

A. The relaxNG grammar and SIDS

The selected grammar is *Relax NG*.[13] The difference between the existing SIDS grammar (a kind of *Mathematica* software programming dialect), and the translation to *Relax NG* is not very important. It is close to a readable grammar and can be easily understood and modified by non-software aware scientists.

We have redefined all the CGNS data structure definitions with *Relax NG*. We tried to be as close as possible to the original SIDS grammar. For example, all type names are the same, node ordering, even if non significant, has been kept. The grammar files have been split in parts in order to be inserted in the SIDS document source. There is a one-to-one relationship between the *SIDS/RelaxNG* document which describe the data structure, and the actual grammar files used by the XML tools. This improves maintenance and consistency.

We had to make some adds and changes to the original SIDS. The actual use of the grammar with tools has imposed some modifications such as defining attributes instead of some nodes used for the node qualification. We have now the `Name` attribute. Many SIDS node names are reserved or recommended, but there was no formal[d] way in SIDS to define a name mandatory or optional. Some missing types were added, such as `DiffusionModel_t` instead of `"int[1+...+IndexDimension]"` in SIDS.

The example below shows the `ZoneIterativeData` sub-tree, which describes the time-dependent data associated to a zone in a CGNS tree. The syntax indicates if the element has a `0:1` cardinality (with "`?`") or a `0:N` (with "`*`"). The "`&`" sign is used to declare a non-ordered list of elements. We do not go into details, the *Relax NG* grammar has been chose because it is public and easy to understand. We developed and checked the grammar using public *java* tools such as *Jing* or *Trang*.

```
ZoneIterativeData_t = element ZoneIterativeData_t
{
        attribute Name { xsd:string "ZoneStepPointers" },
        attribute NumberOfSteps { xsd:unsignedInt },
        attribute DataClass { DataClass_t },
        ( RigidGridMotionPointers_t ?
        & ArbitraryGridMotionPointers_t ?
        & GridCoordinatesPointers_t ?
        & FlowSolutionsPointers_t ?
        & DataArray_t *
        & Descriptor_t *
        & DimensionalUnits_t ?
        & UserDefinedData_t *
        )
}
```

There are two parts in the declaration, the attributes and the elements. An attribute is a value which qualifies the current node, the elements are sub-trees. Some attributes are duplicated in the sub-trees of a given node. For example, a `DataArray` element has its own `Dimension` attribute, which should be the same as its ancestor `Dimension` attribute.

```
...
   attribute IndexDimension { DimensionInteger_t } ,
...
   & DataArray_t         *  # Dimension=IndexDimension
                            # DimensionValues=Constraint(Rind,VertexSize)
...
```

We decided to indicate such attribute dependencies in the comments. This is a lack of the grammar, we cannot define all constraints we would like to have. The use of tools will allow us to check such consistency.

[d] It is done providing a textual comment attached to the node definition.

B. Tree description

A *tree instance* is a *value* of a CGNS tree. In other word, it is a set of values, organized into a tree, which structure is compliant to the CGNS tree definition. This description is an ASCII file in XML syntax. Here's an short example of a tree instance:

```
<CGNSTree CGNSLibraryVersion="2.3" >
<CGNSBase_t Name="D02=Rotor-7A"
            CellDimension="3" PhysicalDimension="3"
            SimulationType="UserDefined" >
  <Zone_t Name="Blade-1"
        CellDimension="3" PhysicalDimension="3"
        ZoneType="Structured" VertexSize="[3,5,7]" CellSize="[2,4,6]"
        VertexSizeBoundary="[0,0,0]">
    <GridCoordinates_t Name="GridCoordinates" IndexDimension="3" VertexSize="[3,5,7]">
      <DataArray_t Name="CoordinateX"
                   DataType="RealDouble" Dimension="3"
                   DimensionValues="[3,5,7]" />
    </GridCoordinates_t>
  </Zone_t>
</CGNSBase_t>
</CGNSTree>
```

You can find the attributes and elements declared in the *SIDS/RelaxNG*. An element starts with a tag with its type name, it has attributes and the sub-trees are enclosed between the *start* tag and the *end* tag.

A textual representation of a CGNS tree allow the creation of *empty patterns*, which can be generated by small and easy to write scripts. These patterns are used for test set generation, for documentation guidelines or examples, or for a partial check of the compliance.

With XML comes the problem of very large data sets[e]. We allow the use of ASCII data sets, but as soon as the data set exceeds a fixed size we use an `URL` to reference an actual binary file. The idea of the SIDS is more in terms of data structure, in terms of a global consistency of a set of data. There is no real constraints on the data contents, except the size and the type of this data.

C. Grammar specialization

The base CGNS grammar can be specialized by the users. The *Relax NG* definitions dedicated to the component is an input of the CGNS/XML compiler. A component may provide only 75% of the CGNS interface. We have to modify the constraints on this interface in order to make sure that input and/or output are compliant to these 75%. In a workflow, you may have a *Structured* solver for the CFD and an *Unstructured* solver for the CSM. However, CGNS allows clever data definitions, that can either be understood by *Structured* and *Unstructured*. This is the goal of the grammar specialization, we want to make sure that component interfaces are tuned for our application.

During the grammar specification phase, a new grammar specialization file can be set as input. This specialization file is a configuration file set by the administrator, or by the tool maintainer. The grammar specialization actually is mandatory in the case of a component interface check. We have to restrict the controls of the data input/output grammar to subset of the CGNS SIDS.

There are some examples of simple restrictions:

- A fixed name for a node

```
ZoneIterativeData_t = element ZoneIterativeData_t
{
        attribute Name { xsd:string "ZoneStepPointers" },
...
```

[e]The current version of SIDS doesn't address this problem. The examples of large CGNS data, such as `Coordinates` or `FlowSolution` are given using a loop syntax without real data.

- A *non-CGNS-mandatory* node suppression (in this example `DataClass` is a comment.)

```
ZoneIterativeData_t = element ZoneIterativeData_t
{
...
#       attribute DataClass { DataClass_t },
...
```

- A restricted enumerate

```
DataClass_t = "Dimensional"
```

We often use names for data qualification. The most common use is a sequence number such as `FlowSolution#5000`, indicating an iteration number for example. The syntaxic pass of the compiler can check such a *name syntax*.

D. Semantics checks

The compiler analysis of the input data goes through several steps. One of the two important stages are the *syntaxic* and the *semantic* stages. The first one if performed when the compiler checks the structure of the tree, regardless of the actual values of *variables*. For example, it checks that `Structured` zone cannot contain a `Elements` node, or that a mandatory `ReferenceState` node is present.

The second stage, the *semantic* stage, checks the *values* and the consistency of these values in the whole data specification. For example it checks that a `PointRange` defining indices of a boundary condition has correct values with respect to the owner zone dimensions.

Some of semantic checks can be added or modified by the user. The *schematron*[14] XML system has been chosen for the semantic rules specification. Other semantic checks are performed by the compiler itself, using *Python* programming language.

The user **cannot** declare his own types in the grammar, because the specialized grammar should be a subset of SIDS, not an extension. For example, we need to declare a `DropletClass` node. We have to use the `UserDefinedData` node, we make sure the node has the expected structure and attributes during the semantic stage. This is an example of a *schematron* rule, used to enforce the presence of specific data in every flow solution (with the error diagnostic):

```
<sch:pattern name="Check FlowSolution">
  <sch:rule context="FlowSolution_t">
  <sch:assert test="DataArray_t[@Name='VelocityX']" diagnostics="E050" />
  <sch:assert test="DataArray_t[@Name='VelocityY']" diagnostics="E051" />
  <sch:assert test="DataArray_t[@Name='VelocityZ']" diagnostics="E052" />
...
<sch:diagnostic id="E050">
   FlowSolution_t {<sch:value-of select="@Name" />} must contain
   a DataArray_t with name "VelocityX"
</sch:diagnostic>
```

Another example, where we want a specific BC to contain both *Neumann* and *Dirichlet* data sets.

```
<sch:rule context="BCDataSet_t">
  <sch:assert test="((@Name='IcingWallData')
              and (count(DirichletData_t) = 1)
              and (count(NeumannData_t) = 1))
              or (@Name!='IcingWallData')"
              diagnostics="E031"/>
</sch:rule>
```

Many complex checks can be performed during the semantic stage. The *schematron* rules language is a bit difficult to use, but it is standard. We avoid to perform very large semantic checks, we think this is more related to the applications. For example, we didn't want to check if numerical choices in `ReferenceState` are relevant regarding the `GorverningEquation` definitions.

E. Use of the compiler

The **C5** compiler is included in the *pyCCCCC Python*[15] package. It uses the *libxml2*[16] XML library, and is released as *Open Source*. We call this tool a compiler because it reads a textual language and it generates a binary representation of the input.

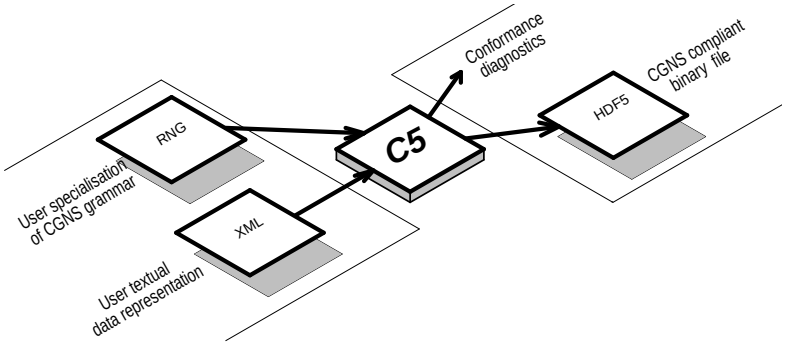

Figure 3. The C5 XML/CGNS compiler files

The usual input is an ASCII representation of a CGNS tree (an XML file), the output is a CGNS binary file (an HDF5 file). The figure 3 shows the input/output process of the compiler.

The *compiler* checks that an ASCII data definition is compliant to a grammar, it also produces binary code in the case of a correct data definition. The user obtains a CGNS binary file which is guaranteed to be CGNS compliant, moreover, this file is compliant to the user restriction of SIDS.

In the case of large data set, the data is located using an *URL* to the actual binary file containing the data. This data is read by **C5**, several binary formats are accepted.

An HDF5 file can be also used as input, for example to get back an XML representation of an existing CGNS tree. But in the case of a non-CGNS compliant file, the performed checks are less robust than using the XML file. If you want to check an HDF5 file, you have to first generate the XML without any check, then you reparse this XML output asking for more checks.

There is a pre-processing phase, using *XSL* filters. The pre-processing is used to set default values, to propagate duplicated attributes (such as `IndexDimension`), to force link creation. This *XSL* tasks are not subject to standardization in the CGNS context, it is a **C5** function.

IV. An application in physics

A. Icing simulation process

Icing is due to caption and freezing of super-cooled water droplets (liquid water droplets at a temperature below the dew point) contained in some clouds which are flown through by aircrafts. When the super-cooled droplets hit the surface of the aircraft, this super- cooled state is broken: the droplets freeze more or less rapidly creating rime or glaze ice depending on temperature values. Without any protection, ice accretion can induce hard aerodynamic penalty by modifying the aerodynamic shape[17] of the wing (for example, horn shape) or can induce extinction of the engine by ingesting blocks of ice in the air entry (often the case for rotorcraft engines). Each year, it causes several incidents or accidents.[18]

The actual challenge is to provide CFD tools[19] which are able to evaluate the penalty due to 3D ice accretion. Ice accretion depends on many parameters such as air quantities (velocity, temperature, pressure) and water quantities (droplet velocity, droplet temperature, liquid water content and median volumic diameter). It also depends on wall quantities such as friction direction, wall heat flux and connective exchange coefficient, and water catch efficiency coefficient. These parameters are computed from an *ice workflow* composed by three kinds of components that will be briefly presented in the next section.

These data structure and definition can be found in the CGNS standard.

B. Components and interfaces

Three components can be identified, and there is a fourth component, the control component which actually drives the simulation. The schema below gives an outline of the components and the interface relationships.

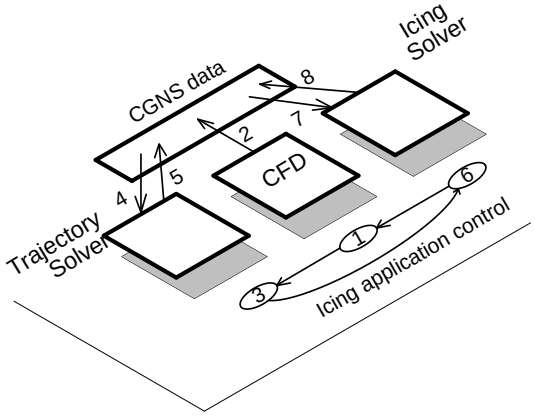

Figure 4. The icing simulation components

The first entry point of this ice workflow is obviously the aerodynamic solver. This component computes air fields such as density, velocity, temperature and pressure. It also computes some wall quantities such as friction direction (for water runback), heat flux and convective exchange coefficient.

The second entry point is the trajectory solver. Generally, the eulerian approach is preferred than the lagrangian one when involving 3D ice accretion. So, this component requires air quantities in the 3D computational domain, given by the aerodynamic solver. It computes water droplet fields such as water droplet velocity and water droplet temperature, and also computes water catch efficiency coefficient.

The last entry point is the ice solver which corresponds to a thermodynamic balance[20] (mass and energy) at the icing wall. This component requires water droplet quantities given by the previous solver, but also wall quantities computed by the first solver. As a result, the thermodynamic solver computes an ice growth rate.

C. The *ice flow* data specification

The specification is an agreement between actors of the ice flow: the component providers/builders and the component users.

Examples of specific requirements of the ice flow are: use of SI units, constraints on structured and unstructured meshes (such as hexa elements only), mandatory data (such as *DropletVelocity*, *FrictionDirection*...). The *generalized* connectivity is required, that is the more general connectivity definition in CGNS and this allow the use by all involved solvers.

The grammar specification is built from the original *relaxNG* grammar for CGNS, it adds restrictions on existing rules. For example, the `GridLocation_t` attribute, which defines the location of the values in the associated data, can be set to force cell centered values for 3D data and to face values for 2D. The requirement leads to the following grammar lines:

```
GridLocation3D_t = "CellCenter"
GridLocation2D_t = "FaceCenter"|"IFaceCenter"|"JFaceCenter"|"KFaceCenter"
```

Some data also have required dimensional units and dimensional exponents. The grammar enforces these definitions.

D. Example of a transient tree

We give here a transient data tree example. This tree is a kind of shuttle tree transferred from one component to the next component. Some of this data can be archived for a subsequent simulation. The tree is compliant to the *ice flow* interfaces grammar.

The CGNS compiler enforces two kind of checks. The first one is the compliance of send/receive data to the CGNS standard and the second is the compliance to the *ice flow* data specification. The tree example[f] in figure 5 shows a CGNS tree containing the solutions at the *predictor* stage. The *corrector* stage is the next stage, it will be added to the data tree with an extra structure, so-called `ZoneIterativeData` in CGNS. This extra structure will point to the data with respect to the step to which they belong. The `DimensionalExponents` node is defined with `DimensionalUnits` for every value. Even if it looks obvious to give units and exponents to a `Density`, no ambiguity is left and this CGNS capability is mandatory in our specification.

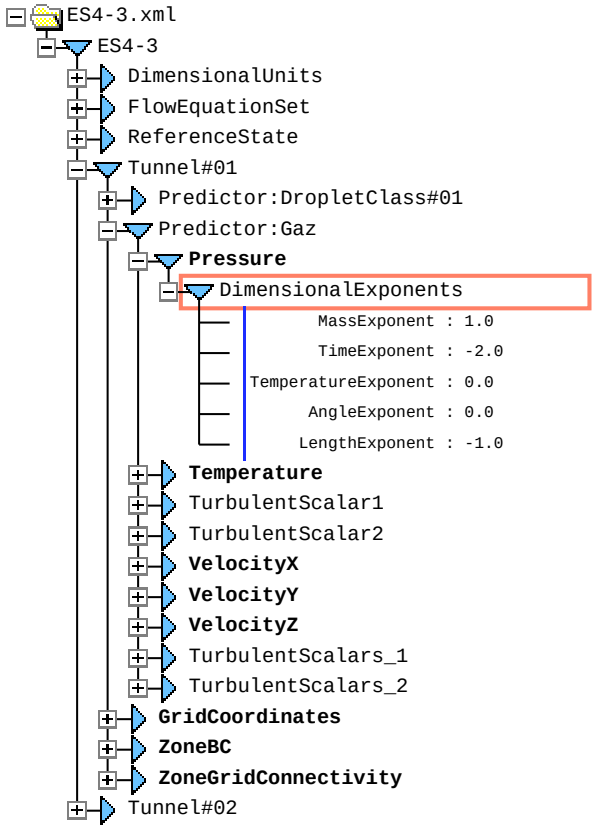

Figure 5. A transient tree example

The CGNS compiler is also used to produce data test sets. The test set is first written in XML, using open source XML tools. The compiler reads the textual data representation and produces a CGNS binary file. All users can edit and modify ASCII XML files, and the compiler is the powerful support to insure a correct data production.

E. Actual impact on team work

The specification of component interface is made at the beginning of the project, even if some extensions or modifications could be done later. Once the interfaces had been defined, the compiler is tuned for these grammars and released to working teams. The users can perform checks and binary generation using the compiler, they cannot modify the grammars anymore.

The benefits of the XML/CGNS compiler involve both the process of the team work, but also the quality of the simulation itself.

The first benefit is the specification phase. The whole simulation system is thought in terms of an architecture, a components and interface definitions. The interface identification and specification help the

[f]The screenshot comes from the **cgt** display tool, it is included in the *pyCCCCC* release.

understanding of dependencies between the solvers, from the data points of view but also from the time (life cycle) point of view. For example, we had to define a *DropletClass* which sets some droplet type for the simulation. For each droplet class we have a *FlowSolution*. Such a need was found during the specification phase, some solvers did require the droplet class, some other didn't. However, we decided to extend the CGNS tree specification with this *DropletClass*, in the case we had to change a component that would require this information[g].

The second benefit is the capability to define the test set without knowledge of the actual component implementations. Actually, this is a full V-cycle process. Some sets of CGNS trees have been created, using *fortran* programs, *python* scripts or even modifying the *XML* textual representation of files and checking them using **C5** to make sure our modifications were CGNS compliant. The test set has been made public to the project teams. They use the files for several purposes:

- Examples for specifying or coding their software component.
- Tests files as input or output verification for their component.
- Data files for feeding component *stubs*.

Another benefit is the capability to define prototype components. Some new ideas can be tested quickly without breaking the whole system. We had to think to extensions, and the whole simulation process was thought in terms of software system as well as a simulation system. For example, we had to add a special boundary condition, the BC data had to be either *input* and *output*. Such a life cycle is not defined in the standard. We have defined our own life cycle for these boundary conditions, reading the our BC data can be performed by all CGNS compliant tools.

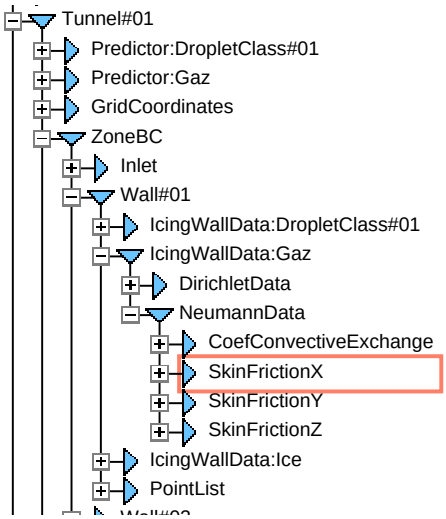

Figure 6. BC data as input/output values

The `SkinFriction` vector usually is a value for CGNS flow solutions. We put this vector in the BC data, as shown in figure 6.

[g]We cannot extend the specification to have the *perfect* interface, allowing all COTS to be connected to our simulation. In that case, the expert have found an obvious dependency which was hidden. The process of defining an interface has show the need for this hidden dependency.

V. Conclusions

The use of the compiler has improved the interface definition and thus the modularity and the interoperability of the simulation workflow. Scientists and engineers can focus on the global system behavior, that is the gist of the actual simulation: the physics. The architecture can be delegated to third party software.

The goal now is to spread this use of the CGNS standard, the use of the compiler itself, and to find agreement between teams working on simulations in the same area, such as *icing*. We will propose the XML mapping to the *CGNS Steering Committee*, with the idea of having a good context for creating a *CGNS compliant label*.

The XML/CGNS compiler will be released as an *Open Source* tool at the end of year 2004.

In Memoriam Robert Henry, ONERA/DMPH/EAG

References

[1]Korel, B., "Black-Box Understanding of COTS Components," *Proceedings of the 7th International Workshop on Program Comprehension*, IEEE Computer Society, 1999, p. 92.

[2]Parker, S., Johnson, C., and Beazley, D., "Computational Steering Software Systems and Strategies," *IEEE Computational Science & Engineering*, Vol. 4, No. 4, /1997, pp. 50–59.

[3]Legensky, S., Edwards, D., Bush, R., Poirier, D., Rumsey, C., Cosner, R., and Towne, C., "CFD General Notation System (CGNS): Status and Future Directions," Aiaa paper 2002-0752, Jan. 2002.

[4]Poirier, D., Allmaras, S., McCarthy, D., Smith, M., , and Enomoto, F., "The CGNS System," Aiaa paper 98-3007, 1998.

[5]Rumsey, C., Poirier, D., Bush, R., and Towne, C., "A User's Guide to CGNS," Nasa/tm-2001-211236, Oct. 2001.

[6]Poinot, M., Rumsey, C., and Mani, M., "Impact of CGNS on CFD workflow," Aiaa paper 2004-2142, 2004.

[7]CGNSTeam, "The CFD General Notation System, Standard Interface Data Structures," Aiaa r-101-2002, Dec. 2002.

[8]Meyers, B. and Chastek, G., "The Use of ASN.1 and XDR for Data Representation in Real-time Distributed Systems," Carnegie mellon univ., software engineering institute, technical report cmu/sei-93-tr-10, Oct. 1993.

[9]W3C, "XML, WWW home page," http://www.w3.org/XML/.

[10]NCSA, *HDF5, WWW home page*, http://hdf.ncsa.uiuc.edu/HDF5.

[11]Wright, W., Gent, R., and Gufond, D., "DRA/NASA/ONERA Collaboration on Icing Research - part II : Prediction of Airfoil ice Accretion," Tech. rep., NASA Technical Report, NASA-CR-202349, 1997.

[12]Vinoski, S., "CORBA: integrating diverse applications within distributed heterogeneous environments," *IEEE Communications Magazine*, Vol. 14, No. 2, 1997.

[13]Clark, J., "RelaxNG, WWW home page," http://www.relaxng.org/.

[14]"Schematron, WWW page," http://www.schematron.com/.

[15]van Rossum, G., "Python, WWW home page," http://www.python.org.

[16]Veillard, D., "The XML C parser and toolkit of Gnome, WWW home page," http://www.xmlsoft.org.

[17]Henry, R., Guffond, D., Aschettino, S., and Duprat, G., "Characterization of Ice Roughness and Influence on Aerodynamic Performance of Simulated ice Shapes," Tech. rep., AIAA Paper 2001-0092, Reno, 2001.

[18]Kind, R., "Icing frost and aircraft flight," *Canadian Aeronautics and Space Journal*, Vol. 4, 1998, pp. 110–118.

[19]Beaugendre, H., *A PDE-Based 3D Approach to In-Flight Ice Accretion*, Ph.D. thesis, McGill University, Montreal, Quebec, 2003.

[20]Messinger, B., "Equilibrium Temperature of an Unheated Icing Surface as a Function of Air Speed," *Journal of the Aeronautical Sciences*, Vol. 20, 1953, pp. 29–42.